nguage designed at Bell Labs. The co
e package have been rewritten and enhanc
s actively supported on the Honeywell T
ersity of Waterloo.

y suited for non-numeric computation
m programming. These usually involve ma
ecisions, computations on integers a
especially characters and bit strings, a
ng point. B programs for such operatio
easier to write and understand than GM

UNIVERSITY OF HOUSTON
UNIVERSITY COMPUTING CENTER
HOUSTON, TEXAS 77004

A TUTORIAL INTRODUCTION TO THE LANGUAGE B

by

B.W. Kernighan
Bell Laboratories
Murray Hi 11r New Jersey

1- Introduction

B is a computer language designed at Bell Labs. The compiler and runtime package have been rewritten and enhanced and the language is actively supported on the Honeywell TSS system at the University of Waterloo.

B is particularly suited for non-numeric computations, typified by system programming. These usually involve many complex logical decisions, computations on integers and fields of words, especially characters and bit strings, and very little floating point. B programs for such operations are substantially easier to write and understand than GMAP programs. The generated code is quite good. Implementation of TSS subsystems is an especially good use for B.

B is reminiscent of BCPL C2J, for those who can remember. The original design and implementation are the work of K. L. Thompson and D. M. Ritchie, their original Honeywell 6070 version was substantially improved by S. C. Johnson, who also wrote the runtime library. The compiler was rewritten and enhanced by R. Braga and the runtime package was rewritten by T. J. Thompson, both of the University of Waterloo.

This memo is a tutorial to make learning B as painless as possible. Most of the features of the language are mentioned in passing, but only the most important are stressed. This memo has been extensively revised by the University of Waterloo to suit the local runtime environment. Users who would like the full story should consult the "B Reference Manual" , by R. P. Gurd, which should be read for details anyway.

We will assume that the user is familiar with the mysteries of TSS, such as creating files, text editing, and the like, and has programmed in some language before. Throughout, the symbolism (→ n) implies that a topic will be expanded upon in section n of this manual. Appendix E is an index to the contents.

Throughout, the symbolism (→ n) implies that a topic will be expanded upon in section n of this manual. Appendix E is an index to the contents.

2. General Layout of B Programs

```
1       main() {
2               -- statements --
3       }
4
5       newfunc(arg1,arg2) {
6               -- statements --
7       }
8
9       fun3(arg) {
10              -- more statements --
11      }
```

All B programs consist of one or more "functions", which are similar to the functions and subroutines of a Fortran program, or the procedures of PL/I. main is such a function, and in fact all B programs must have a main. Execution of the program begins at the first statement of main, and usually ends at the last, main wilt usually invoke other functions to perform its job, some coming from the same program, and others from libraries. As in Fortran one method of communicating data between functions is by arguments. The parentheses following the function name surround the argument list; here main is a function of no arguments, indicated by `()`. The `{}` enclose the statements of the function.

Functions are totally independent as far as the compiler is concerned, and main need not be the first, although execution starts with it. This program has two other functions: `newfunc` has two arguments, and `fun3` has one. Each function consists of one or more statements which express operations on variables and transfer of control. Functions may be used recursively at little cost in time or space (→ 31).

Most statements contain expressions, which in turn are made up of operators, names, and constants, with parentheses to alter the order of evaluation, B has a particularly rich set of operators and expressions, and relatively few statement types.

The format of B programs is quite free: we will strive for a uniform style as a good programming practice. Statements can be broken into more than one line at any reasonable point, i.e., between names, operators, and constants. Conversely, more than one statement can occur on one line. Statements are separated by semicolons. A group of statements may be grouped together and treated as a single statement by enclosing them in `{}`; in this case, no semicolon is used after the `'}'` . This convention is used to lump together the statements of a function.

3. Compiling and Running B Programs

To compile B programs, one uses the B command, which creates a temporary executable load module in a file called ".h". The command "go" can then be used to execute this load module. Thus, assuming "bsource" contains the complete b source program:

```
*b bsource
* go
```

Of course, in many cases, there will be numerous lines of output between the "b" command and the next '*' prompt, informing you of the things the compiler disliked about your program. In this case, the "go" will obviously be a little premature. (→ Appendix A : B Compiler Diagnostics). If you want to save your errors for reference,

```
* b bsource > filename
```

will send them to the file "filename".

4. A Working B Program; Variables

```
1       main() {
2               auto a, b, c, sum;
3               a = 1 ; b = 2; c = 3;
4               sum = a + b + c;
5               putnumb(sum);
6       }
```

This simple program adds three numbers, and prints the sum. It illustrates the use of variables and constants, declarations a bit of arithmetics and a function call. Let us discuss them in reverse order.

`putnumb` is a library function of one argument, which will print a number on the terminal (unless some other destination is specified (→ 29)). The code for a version of `putnumb` can be found in (→ 31). Notice that a function is invoked by naming it, followed by the argument(s) in parentheses. There is no CALL statement as in Fortran.

The arithmetic and the assignment statements are much the same as in Fortran except for the semicolons. Notice that we can put several on a line if we want. Conversely, we can split a statement among lines if it seems desirable - the split may be between any of the operators or variables but not in the middle of a variable name.

One major difference between B and Fortran is that B is a typeless language: there is no analog in B of the Fortran `IJKLMN` convention. Thus `a, b, c` and `sum` are all 36-bit quantities, and arithmetic operations are integer. This is discussed at length in the next section (→ 5).

Variable names have one to eight ascii characters, chosen from `A-Z`, `a-z`, `.`, `_`, `0-9`, and start with a non-digit. Stylistically, it's much better to use only a single case (upper or lower) and give functions and external variables (→ 7) names that are unique in the first six characters. (Function and external variable names are used by batch GMAP, which is limited to six character single-case identifiers.)

The statement `"auto..."` is a declaration, that is, it defines the variables to be used within the function. auto in particular is used to declare local variables, variables which exist only within their own function (`main` in this case). Variables may be distributed among `auto` declarations arbitrarily, but all declarations must precede executable statements. All variables must be declared, as one of `auto`, `extrn` (→ 7), or implicitly as function arguments (→ 8).

`auto` variables are different from Fortran local variables in one important respect - they appear when the function is entered, and disappear when it is left. They cannot be initialized at compile time, and they have no memory from one call to the next.

5. Arithmetic; Octal Numbers

Normal arithmetic in B is `integer`, unless special functions are written. There is no equivalent of the Fortran `IJKLMN` convention, no special recognition of floating-point (although see the "B Reference Manual" for details of how it can be done), no data types, no type conversions, and no type checking. Although it is possible to do floating-point arithmetic in B, it is neither very elegant nor very efficient. But then this is not what B was originally designed to be able to do. Users of double-precision complex will have to fend for themselves. (But see → 33 for how to call Fortran subroutines.)

The arithmetic operators are the usual `'+'`, `'-'`, `'*'` and `'/'` integer division). But notice that there is no equivalent of Fortran's `'**'` for exponentiation. But we do have a remainder operator `'%'`:

```
x = a % b
```

sets `x` to the remainder after `a` is divided by `b` (both of which should be positive).

Since B is often used for system programming and bit-manipulation, octal numbers are an important part of the language. The syntax of B says that any number that begins with `0` is an octal number (and hence can't have any 8's or 9's in it). Thus `0777` is an octal constant, with decimal value `511`.

6. Characters; putchar; Newline

```
1       main() {
2               auto a;
3               a ='b';
4               putchar(a);
5               putchar('*n');
6       }
```

This program prints `"b"` on the terminal: it illustrates the use of character constants and variables. A "character" is one to four ascii characters, enclosed in single quotes. The characters are stored in a single machine word, right-justified and zero-filled. we used a variable to hold `"b"`, but could have used a constant directly as well. putchar is a library function which takes a single right-justified "character" as an argument, and sends it to the current output unit (→ 29).

The sequence `"*n"` is B jargon for "newline character", which, when printed, skips the terminal to the beginning of the next line. No output occurs until putchar encounters a `"*n"` or the program terminates or desires input. Omission of the `"*n"`, a common error, causes all output to appear on one line, which is usually not the desired result. There are several other "escapes" like `"*n"` for representing hard-to-get characters (→ Appendix B). Notice that `"*n"` is a single character. If you are ever unfortunate to say `"putcharl('*')"` in your program, meaning to print a single asterisk, you will find the B compiler gets rather confused and unsympathetic, since represents a single quote. Simply correcting it to say `"putchart('**')"` will remove pages of error messages.

Since B is a typeless language, arithmetic on characters is quite legal, and even makes sense sometimes:

```
c = c + 'A' - 'a'
```

converts a single character stored in `c` to (making use of the fact that corresponding ascii a fixed distance apart).

7. External Variables

```
1       main() {
2               extrn a, b, c;
3               putchar(a); putchar(b); putchar(c);
4               putchar('*n');
5       }
6       a 'h';
7       b 'i';
8       c '!';
```

This example Illustrates external variables, variables which are rather like Fortran `COMMON`, in that they exist external to all functions, and are (potentially) available to all functions. Any function that wishes to access an external variable must contain an `extrn` declaration for it. Furthermore, we must define all external variables outside any function. For our example variables `a`, `b`, and `c`, this is done in the last three lines.

External storage is static, and always remains in existence, so it can be initialized. (Conversely, auto storage appears whenever the function is invoked, and then disappears when the function is left. Auto variables of the same name in different functions are unrelated.) initialization is done as shown: the initial value appears after the name, as a constant. We have shown character constants; other forms are also possible. External variables are initialized to zero if not set explicitly.

8. Functions

```
1       main() {
2               extrn a, b, c, d;
3               put2char(a,b);
4               put2char(c,d);
5       }
6
7       put2char(x,y) {
8               putchar(x);
9               putchar(y);
10      }
11
12      a 'h'; b 'i'; c '!'; d '*n';
```

This example does the same thing as the last, but uses a separate (and quite artificial) function of two arguments, `put2char`. The dummy names `x` and `y` refer to the arguments throughout `put2char`; they are not declared in an `auto` statement. We could equally well have made `a`, `b`, `c`, `d` auto variables, or have called `put2char` as

```
        put2char('h','i');
        etc.
```

Execution of `put2char` terminates when it runs into the closing `'}'`, just as in main, control returns to the next statement of the calling program.

9. Input; getchar

```
1       main() {
2               auto c;
3               c = getchar0;
4               putchar(c);
5       }
```

`getchar` is another library IO function, It fetches one character from the input line each time it is called, and returns that character, right adjusted and zero filled, as the value of the function. The actual implementation of `getchar` is to read an entire line, and then hand out the characters one at a time, After the `'*n'` at the end of the line has been returned, the next call to `getchar` will cause an entire new line to be read, and its first character returned. Input is generally from the terminal (→ 29).

10. Relational Operators

==	equal to (".EQ." to Fortraners)
!=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

These are the relational operators; we will use `'!='` ("not equal to") in the next example. Don't forget that the equality test is using just one `'='` means assignment, and leads to disaster of one sort or another. Note: all comparisons are arithmetic, not logical. It is also important to realize that, since B is a typeless language, the result obtained by evaluating a relational operation is treated the same as an arithmetic result. The relational operators are defined such that the expression has a value of one if the relation between its two operands is satisfied, and zero if it is not.

11. if; goto; Labels; Comments

```
1       main() {
2               auto c;
3       read:
4               c = getchar(c);
5               putchar(c);
6               if(c != '*n') goto read; /* loop if not a newline */
7       }
```

This function reads and writes one line. It illustrates several new things. First and easiest is the comment, which is anything enclosed in `/*...*/`, just as in PL/I.

read is a label - a name followed by a colon `':'`, also just as in PL/I. A statement can have several labels if desired. (Good programming practice dictates using few labels, so later examples will strive to get rid of them.) The `goto` references a label in the same function.

The `if` statement is one of several conditionals in B. (`while` (→ 14), `switch` (→ 2, and `'?:'` (→ 1 are some of the others. There is also the `do-while`, and the `for`, but we are setting them aside for the moment. They are discussed fully in the "B Reference Manual".) Its general form is

```
if(expression) statement
```

The condition to be tested is any expression enclosed in parentheses. It is followed by a statement. The expression is evaluated, and if its value is `non-zero`, the statement is executed. In practice this usually means you are testing to see whether a relational expression is satisfied (remembering the way the relational operators return a zero or one value (→ 10)), as we have done above.

One of the nice things about B is that the statement can be made arbitrarily complex (although we've used very simple ones here) by enclosing simple statements in `{}`. (→ 12)

12. Compound Statements

```
1       if(a<b) {
2               t = a;
3               a = b;
4               b = t;
5       }
```

As a simple example of compound statements, suppose we want to ensure that `a` exceeds `b`, as part of a sort routine. The interchange of `a` and `b` takes three statements in B, they are grouped together by `{}` in the example above.

As a general rule in B, anywhere you can use a simple statement, you can use any compound statement, which is just a number of simple or compound ones enclosed in `{}`. Compound statements are not followed by semicolons.

The ability to replace single statements by complex ones at will is one thing that makes B much more pleasant to use than Fortran. Logic which requires several `GOTO`'s and labels in Fortran can be done in a simple clear natural way using the compound statements of B.

13. Function Nesting

```
read:
        c = putchar(getchar());
        if(c != '*n') goto read;
```

Functions which return values (as all do, potentially → 24) can be used anywhere you might normally use an expression. Thus we can use `getchar` as the argument of `putchar`, nesting the functions. `getchar` returns exactly the character that `putchar` needs as an argument, `putchar` also passes its argument

back as a value, so we assign this to `c` for later reference.

Even better, if `c` won't be needed later, we can say

```
read:
    if(putchar(getchar()) != '*n') goto read;
```

14. While statement; Assignment within an Expression

```
while((c = getchar()) != '*n') putchar(c);
```

Avoiding goto's is frequently good practice, and here is a similar example without `goto`'s. This has almost the same meaning as the previous one. (The difference? This one doesn't print the terminating `'*n'`.) The `while` statement is basically a loop statement, whose general form is

```
while(expression) statement
```

As in the if statement, the expression and the statement can both be arbitrarily complicated, although we haven't seen that yet. The action of while is

```
(a) evaluate the expression
(b) if its value is not zero (i.e. "true"),
     do the statement and return to (a)
```

Our example gets the character and assigns it to `c`, and tests to see if it's a `'*n'`. If it isn't, it does the statement part of the `while`, a `putchar`, and then repeats.

Notice that we used an assignment statement `"c = getchar()"` within an expression. This is a handy notational shortcut, usually produces better code, and is sometimes necessary if a statement is to be compressed. This works because an assignment statement has a value, just as any other expression does. This also implies that we can successfully use multiple assignments like

```
x = y = f(a) + g(b)
```

As an aside, the extra pair of parentheses in the assignment statement within the conditional were really necessary: if we had said

```
c = getchar() != '*n'
```

`c` would be set to `0` or `1` depending on whether the character-fetched was a newline or not. This is because the binding strength of the assignment operator `'='` is lower than the relational operator `'!='`. (→ Appendix D).

15. More on whiles; Null Statement; exit

```
1       main() {
2               auto c;
3               while(1) {
4                       while((c = getchar()) != ' ')
5                               if(putchar(c) == '*n') exit();
6                       putchar('*n');
7                       while((c = getchar()) == ' ');  /* skip blanks */
8                       if(putchar(c) == '*n') exit();
9                                               /* done when newline */
10              }
11      }
```

This terse example reads one line from the terminal, and prints each `non-blank` string of characters on a separate line: one or more blanks are converted into a single new-line.

Line four fetches a character, and if it is non-blank, prints it, and uses the library function exit to terminate execution if it's a newline. Note the use of the assignment within an expression to save the value of `c`.

Line 7 skips over multiple blanks. It fetches a character and if it's blank, does the statement `';'`. This is called a `null` statement - it really does nothing. Line 7 is just a very tight loop, with all the work done in the expression part of the `while` clause.

The slightly odd-looking construction

```
while(1) {
        ----
}
```

is a way to get an infinite loop. `"1"` is always `non-zero`, so the statement part (compound here!) will always be done. The only way to get out of the loop is to execute a `goto` (→ 11) or `break` (→ 26), or `return` from the function with a `return` statement (→ 24), or terminate execution by `exit`.

16. Else Clause; Conditional Expressions

```
if (expression)
        statementl
else
        statement2
```

is the most general form of `if` - the `else` part is sometimes useful. The canonical example sets `x` to the minimum of `a` and `b`:

```
if(a>b)
        x =a;
else
        x = b;
```

If's and else's can be used to construct logic that branches one of several ways, and then rejoins, a common programming structure, in this way:

```
if(---) {
        ---
        ---
} else if (---) {
        ---
        ---
} else if (---) {
        ---
        ---
}
etc.
```

The conditions are tested in order, and exactly one block is executed - the first one whose if is satisfied. When this block is finished, the next statement executed is the one after the last "else if" block.

In spite of the logical evaluation of constructs like the above tangle of if's and else's, they are usually difficult to for people to follow, and can frequently be avoided by using B's switch statement (→ 26), which is a multi-way conditional branch.

B provides an alternate form of conditional which is often more concise and efficient. It is called the "conditional expression" because it is a conditional which actually has a value and can be used anywhere an expression can. A statement to set x to the minimum of a and b can best be expressed as

```
x = a < b ? a  : b;
```

The meaning is: evaluate the expression to the left of '?'. If it is true, evaluate the expression between '?' and ':' and assign that value to x. If it's false, evaluate the expression to the right of and assign its value to x.

17. Unary Operators

```
auto k;
k = 0;
while (getchart() != '*n') ++k;
```

B has several interesting unary operators in addition to the traditional '-'. The program above uses the increment operator '++' to count the number of characters in an input line. "++k" is equivalent to "k=k+1" but generates better code. '++' may also be used after a variable, as in

```
k++
```

The only difference is this. Suppose k is 5. Then

```
x = ++k
```

sets `k` to `6` and then sets `x` to `k`, i.e., to `6`. But

```
x = k++
```

sets `x` to `5`, and then `k` to `6`.

Similarly, the unary decrement operator can be used either before or after a variable to decrement its value by one.

18. Bit Operators

```
c = 0;
i = 36;
while((i = i-9) >= 0) c = c | getchar() << i;
```

B has several operators for logical bit-manipulation. The example above illustrates two, the `OR` `'|'`, and the left shift `'<<'`. We're not going to explain bit-manipulation here. If you really don't know what we're talking about, you'd better look the terms up somewhere else. The above code packs the next four 9-bit characters from the input into the single 36-bit word `c`, by successively left-shifting the character by i bit positions to the correct position and `OR`-ing it into the word.

The other logical operators are `AND '&'`, logical right shift `'>>'`, exclusive `OR '^'`, and 1's complement `'~'`.

```
x = ~y&0777
```

sets `x` to the last 9 bits of the 1's complement of `y`, and

```
x = x&077
```

replaces `x` by its last six bits.

The order of evaluation of unparenthesized expressions is determined by the relative binding strengths of the operators involved (→ Appendix D).

19. Assignment Operators

An unusual feature of B is that the normal binary operators like `'+'`, `'-'`, etc. can be combined with the assignment operator `=` to form new assignment operators. For example,

```
x =- 10
```

means `"x=x-10"`. `'-='` is an assignment operator. This convention is a useful notational shorthand, and usually makes it possible for B to generate better code.

Because assignment operators have lower binding strength than any other operator, the order of evaluation should be watched carefully:

```
x = x<<y|z
```

means "shift `x` left `y` places, then `OR` in `z`, and store in `x`". But

```
x <<= y|z
```

means "shift `x` left by `y|z` places", which is rather different.

The character-packing example can now be written better as

```
c= 0;
i = 36;
while((i -= 9) >= 0) c =| getchar()<<i;
```

20. Vectors

A vector is a set of consecutive words, somewhat like a one-dimensional array in Fortran. The declaration

```
auto v[10];
```

allocates 11 consecutive words? `v[0]`, `v[1]`, ..., `v[10]` for `v`. Reference to the `i`-th element is made by `v[i]`; the `[]` brackets indicate subscripting. (There is no Fortran-like ambiguity between subscripts and function call.)

```
sum = o;
i = 0;
while(i<=n) sum =+ v[i++];
```

This code sums the elements of the vector `v`. Notice the increment within the subscript - this is common usage in B. Similarly, to find the first zero element of `v`,

```
i = -1;
while(v[++i]);
```

21. External Vectors

Vectors can of course be external variables rather than `auto`. We declare `v` external within the function (don't mention a size) by

```
extrn v;
```

and then outside all functions write

```
v[10];
```

External vectors may be initialized just as external simple variables:

```
v[10] 'hi!', 1, 2, 3, 0777;
```

sets `v[0]` to the character constant `'hi!'`, and `v[1]` through `v[4]` to various numbers. `v[5]` through `v[10]` are not initialized.

22. Addressing

To understand all the ramifications of vectors, a digression into addressing is necessary. The actual implementation of a vector, for example the vector `"v"` above, consists of a word, (which is what the symbol `"v"` actually refers to) which contains in its lower half the address of the `0`-th word of the vector; `v` is correctly termed a `"pointer"` to the actual vector, since you look at the actual word `v` to find out where the assigned storage for the vector is. This fits in with the fact that B is typeless; the contents of `v` can be assigned to any other variable, and then that variable can, in fact, be subscripted, since it will reference the same storage allocated for the vector "pointed to" by `v`. This is allowed because the `[]` subscripts are actually implemented as binary operators, which don't worry about what their operands are. This will become clear below.

This implementation can lead to a fine trap for the unwary. If we say

```
auto u[10], v[10];
---
u = v;
v[0] = ...
v[1] = ...
---
```

the unsuspecting might believe that `u[0]` and `u[1]` contain the old values of `v[0]` and `v[1]`; in fact, since `u` is just a `pointer` to `v[0]` , `u` actually refers to the new content. The statement `"u = v"` does not cause any copy of information into the elements of `u`; they may indeed be lost because `u` no longer points to them.

The fact that vector elements are referenced by a pointer to the zeroeth entry makes subscripting trivial - the address of `v[i]` is simply `v+i`. Furthermore, constructs like `v[i][j]` are quite legal: `v[i]` is simply a pointer to a vector, and `v[i][j]` is its j-th entry. You have to set up your own storage, though.

To facilitate direct manipulation of addresses when it seems advisable, B provides two unary address operators, `'*'` and `'&'`. `'&'` is the address operator so `"&x"` is the address of `x`, assuming it has one. `'*'` is the indirection operator;

`"*x"` means use the content of `x` as an address. Don't get upset by the fact that these operators look exactly like the ones you learned for multiplication and bitwise-and, those interpretations apply only when the symbols are used as binary operators. Context makes it quite clear to both the compiler and people reading the program whether the symbol is being used as a binary or unary operator, and, if it is unary, the address interpretation applies.

Subscripting operations can now be expressed differently:

```
x[0]     is equivalent to *x
x[1]     is equivalent to *(x+1)
&x[0]    is equivalent to x
```

and

```
v[i][j]  is equivalent to *(*(v+i)+j)
```

23. Strings

A string in B is in essence a vector of characters. It is written as

```
"this is a string"
```

and, in internal representation, is a vector with characters packed four to a word, left justified, and terminated by a mysterious character known as `'*e'` (ascii NUL = 000). Thus the string

```
"hello"
```

has six characters, the last of which is `'*e'`. Characters are numbered from zero. Notice the difference between strings and character constants. Strings are double-quoted, and have zero or more characters; they are left justified and terminated by an explicit delimiter, `'*e'`. Character constants are single-quoted, have from one to four characters, and are right justified and zero-filled.

Some characters can only be put into strings by "escape sequences" (→ Appendix B). For instance, to get a double quote into a string, use `'*"'`, as in

```
"He said, *"Let's go.*""
```

External variables may be initialized to string values simply by following their names by strings, exactly as for other constants. These definitions initialize `a`, `b`, and three elements of `v`:

```
a "hello"; b "world'*;
v[2] "now is the time", "for all good men",
    "to come to the aid of the party";
```

Obviously, though, since all variables are a single word, the variables aren't initialized directly as strings. As is

the case with any vectors (→ 22) the variable itself is really a pointer to the start of storage occupied by the string. B provides several library functions for manipulating strings and accessing characters within them in a relatively machine-independent way. These are all discussed in more detail in section 7.5 of the "B Reference Manual". The two most commonly used are char and lchar. char(s,n) returns the n-th character of s, right justified and zero-filled. lchar(s,n,c) replaces the n-th character of s by c, and returns c.

(The code for char is given in the next section.)

Because strings are vectors, and the actual variable itself is just a pointer, writing "s1=s2" does not copy the characters in s2 into s1, but merely makes s1 point to s2. To copy s2 into s1, we can use char and lchar in a function like strcopy:

```
strcopy(s1,s2) {
        auto i;
        i = 0;
        while(lchar(s1,i,char(s2,i)) != '*e') i++;
}
```

24. Return statement

How do we arrange that a function like char returns a value to its caller? Here is char:

```
char(s,n) {
        auto y,sh,cpos;
        y = s[n/4];             /* word containing n-th char */
        cpos = n%4;             /* position of char in word */
        sh = 27-9*cpos;         /* bit positions to shift */
        y =  (y>>sh)&0777;      /* shift and mask all but 9 bits */
        return(y);              /* return character to caller */
}
```

A return in a function causes an immediate return to the calling program. If the return is followed by an expression in parentheses, this expression is evaluated, and the value returned to the caller. The expression can be arbitrarily complex, of course: char is actually defined as

```
char(s,n) return((s[n/4]>>(27-9*(n%4)))&0777);
```

Notice that char is written without {}, because it can be expressed as a simple statement.

25. Other String Functions

concat(s,a1,a2,...,a10) concatenates the zero or more strings ai into one big string s, and returns s (which means a pointer to s[0]).

`s` must be declared big enough to hold the result (including the terminating `'*e'`).

`getstr(s)` fetches the entire next line from the input unit into `s`, replacing the terminating `'*n'` by `'*e'`. In B, it is

```
getstr(s){
        auto c,i;
        while((c=getchar()) != '*n') lchar(s,i++,c);
        lchar(s,i,'*e');
        return(s) ;
}
```

Notice that `getstr` just assumes that it is passed an argument pointing to a free area of storage, It has no way of checking! This means that if you call `getstr`, the argument must have either been declared a vector of sufficient size, or have been made to point to such an area. Failure to do this is a common mistake of beginning B programmers who have not programmed much on a lower level before, and perhaps misinterpret the meaning of the notion "typeless". Suppose you call `getstr` with an "ordinary" variable. This variable probably contains some value. `"getstr"` will simply take that value as an address, and, as long as that address is within the user's allotted memory, will happily stomp all over the words beginning at that address. This is one of the things that makes B so much fun to use.

Similarly `putstr(s)` puts the string `s` onto the output unit, except for the final `'*e'`. Thus

```
auto s[20] ;
putstr(getstr(s)); putchar('*n');
```

copies an input line to the output.

26. Switch Statement; break

```
1       loop:
2               switch(c=getchar()) {
3
4       pr:
5               case 'p':       /* print */
6               case 'P':
7                       print();
8                       goto loop;
9
10              case 's':       /* subs */
12              case 'S':
13                      subs() ;
14                      goto pr;
15
16              case 'd':       /* delete */
17              case 'D':
18                      delete();
19                      goto loop;
20
21              case '*n':      /* quit on newline */
22                      break;
23
24              default:
25                      error(); /* error if fall out */
26                      goto loop;
27
28              }
29              ---
```

This fragment of a text-editor program illustrates the switch statement, which is a multi-way branch. Here, the branch is controlled by a character read into `c`. If `c` is `'p'` or `'P'`, function print is invoked. If `c` is an `'s'` or `'S'`, `subs` and `print` are both invoked. Similar actions take place for other values. If `c` does not mention any of the values mentioned in the case statements, the default statement (in this case, an `error` function) is invoked.

The general form of a switch is

```
switch(expression) statement
```

The statement is always complex, and contains statements of the form

```
case constant-expression:
```

The constants in our example are characters, but they could also be numbers. The expression is evaluated, and its value matched against the possible cases. If a match is found, execution continues at that `case`. (Execution may of course fall through from one `case` to the next, and may transfer around within the `switch`, or transfer outside it.) Notice we put multiple cases on several statements.

If no match is found, execution continues at the statement labeled "default:" if it exists; if there is no match and no default, the entire statement part of the switch is skipped.

The break statement forces an immediate transfer to the next statement after the switch statement, i.e., the statement after the {} which enclose the statement part of the switch. break may also be used in while statements in an analogous manner. break is a useful way to avoid useless labels.

27. Logical Operators

B has two logical operators, '&&' and '||'designed for use in compound relational expressions.

'&&', or logical 'and', returns a one if both of its operands are non-zero, and a zero otherwise.

'||', or logical 'or', returns a zero if both of its operands are zero, and a one otherwise.

I.e. these operators behave just the way you would expect, if you consider non-zero to be 'true', and zero to be 'false'. In most cases, this has the same effect as using the bitwise '&' and '|' operators, since the relational operators ('!=' and co.) return either a one or a zero. However, it permits the unambiguous use of 'flags' (which are essentially words used like Fortran logical variables), since the operand need only be non-zero to be 'true'. It also guarantees left-to-right evaluation of the logical expressions (allowing the effects of side-effects to be determined in an implementation-independent manner.).

As an example, consider the following simple-minded getnum function for reading in unsigned (implicitly positive) decimal integers. This function will read in characters until either it has found a non-numeric character, or d digits have been read, where d is passed as an argument by the calling function. It then returns as its value the value of the decimal number represented by the characters. It determines the value of the digits by subtracting from each character the value '0', leaving the value of the individual digit behind, possible because of the way the digits are arranged in ascii.

```
getnum(d) {
        auto c,n;
        n = 0;
        while((c=getchar()) >= '0' && c <= '9' && d--)
                n = n*10 + c - '0';
        return (n);
}
```

Note the use of "d--". This is common practice among B programmers, who hate to explicitly relate anything to zero. This works because of the definition of '&&' and '||', as well as the very definition of the control statements. When first learning B you may often find it helpful to imagine all the "!= 0"'s as being there, though. Note also that when we compare c to '9', we are using the new value (returned from getchar()).

In addition to `&&` and`||`, there is a logical `'not'` operator `'!'`, which is a unary operator returning a result of one if the operand it is applied to is zero, and non-zero otherwise.

For example:

```
!( a == b )
```

i s equi valent to:

```
a != b
```

One of its most common uses is to avoid saying `"== 0"`, as in

```
if(!a) statement
```

this is especially meaningful in the case of some function return values and flags.

28. Formatted IO

In this section, we point to the function `printf`, which is used for producing formatted IO. This function is so useful that it is actually placed in Appendix C for ready reference.

29. Lots More on IO

Basically, all IO in B is character-oriented, and based on `getchar` and `putchar`. By default, these functions refer to the standard input, and standard output, unless explicit steps are taken to change it. For "quicky" B programs, there is an interesting feature of the run-time support package that lets you easily divert the IO, by means of the command line used to start the program. That is, "standard output" normally means the terminal, but if

```
<filename
```

appeared on the command line, standard input is instead, from file "filename". Similarly

```
>filename
```

will divert the standard output.

At any time, a B program has one default input unit, and one
default output unit, for use by `getchar` and `putchar`. The
unit is represented by a number between `-5` and `10` (rather like Fortran file-numbers). Units `-5` to `-1` have special predefined functions, but the rest are available to be associated with the user's files. When execution of a B program begins, the input unit is `0`, which means `"read from the standard input"`, and the output unit is `1`, meaning `"write to the standard output"`. Unless redirected by means of `"<"` or `">"` standard input and output are both the terminal.

MOST IO functions will accept a unit number as an optional argument, and use that unit to perform their function, instead of the standard input or output. For example:

```
printf(4*"banana*n");
```

will send `"banana*n"` to the file associated with unit `4`. The explain files for the individual functions involved (`expl b lib` function) should be consulted to determine exactly which functions accept units how.

Of course, we must do something to associate the unit number with a file. (In case you're trying to figure out how `printf` knows that a unit number is a unit number and not a pointer to a string, it just says that numbers in the range `0` to `10` cannot be pointers, which is very reasonable, since if you ever put strings at those locations, your program would probably very shortly meet with disaster !). Anyway B requires no ugly IBM JCL to be run before the program to initialize these unit numbers; instead the initialization is all done within the program, by calling the standard library function open, thusly:

```
unit = open(filename,mode);
```

where `filename` is a string containing the `name` of the file to be opened, and `mode` is a string telling open what you intend to do with the file. `open` will return as its value the new unit number you have “opened", and so you should save it in a variable (unit, in this case). Very often it is nice to read the command line to find out what file the user of your program wants to use (→ 33). (If you have some reason for wanting to use a certain particular unit number, there is a way you can specify it, but usually this is neither necessary nor useful.).

To do input from a file, we set rd.unit to a value, and assign a file name to that unit. Then, until the assignment or the value is changed, getchar reads its characters from that file. Thus, to open for reading an ascii permanent file called "cat/file", as input unit 5,

For example:

```
filein = open("fbaggins/ring","r");
```

will allow you to read from the file `"fbaggins/ring"`, assuming it exists. File names containing a `'/'` are assumed to be permanent files; if you try to open a permanent file which is already accessed, it is released and reaccessed. If it can't be found, the access fails. A file name without `'/'` may or may not be a permanent file, so if a file of this name is already accessed, it is used without question. If it is not already accessed, a search of the user's catalog is made for it. If it can’t be found there either, and writing is desired, a temporary file is made.

An important point should be noted regarding open, however. After each call to open, the default unit is actually changed. That is, the next call to `getchar`, will now return the first character in `"fbaggins/ring"`. (When the end-of- file is reached, all successive calls to `getchar` will return `'*e'`). Often this change of the default unit is very convenient, but at other times, it can be a pain.

The value of the current default input unit can be determined by calling the library function `.read`:

```
where = .read();
```

will assign the current default unit to the variable `"where"`. A similar function, `.write`, exists in the B library for determining the current default output unit. (A file is opened for output and associated with a unit number by using `"w"` as the mode string when calling open).

Both `.read` and `.write` take an optional argument, which makes them much more useful, since the argument will be used as the new value for the default `input` or `output` unit. That is:

```
oldunit = .write(A);
```

will save the current default output unit in `"oldunit"`, but then make 4 the default, so that

```
putchar('*n');
```

will send a newline character to unit 4. Of course, if you already know what the current default is, you could just say:

```
.write(4);
```

to merely switch the unit.

`.read` and `.write` are useful if you have to do a lot of `IO` with a unit that it wasn't convenient to open last, since it avoids specifying the unit number all the time. (And, indeed, some functions will deal only with the default units).

All units are closed automatically upon program termination. That is, any output for output files is flushed and end of file is written; any input for input files is thrown away. There is a library function `"close"`, though if you want to close some units yourself before the program terminates. You simply say;

```
close(unit);
```

where "unit" contains the number of the unit to be closed.

As a concrete example of file `IO` in B, consider the following function `fcopy`, which will copy one file to another. Its two arguments, `f1` and `f2` are assumed to be strings containing the names of the input and output files, respectively.

```
1       fcopy(f1,f2) {
2               /* copy file f1 into f2 */
3
4               auto fi1,fi2, savein, saveout;
5               savein = .read();         /* save default input unit */
6               saveout = .write();       /* save default output unit */
7               fi1 = open(f1, "r");      /* open f1 for reading */
8               fi2 = open(f2, "w") ;      /* open f2 for writing */
9
10              while(putchar(getchar()) != '*e')
11                      ;
12                      /* copy default input to default output */
13              close(fi1);       /* tidy up a bit */
14              close(fi2);
15              .read(savein);    /* restore caller's default units */
16              .write(saveout);
17      }
```

30. System Call from TSS(Time-Sharing System)

TSS users can easily execute other TSS subsystems from a running B program, by using the library function SYSTEM. For instance,

```
system("jrun file");
```

acts exactly as if we had typed

```
jrun file
```

in response to the command level '*' prompt. The argument of `SYSTEM` is a string, which is executed as if typed at command level. (No `'*n'` is needed at the end of the string). Return is to the B program unless disaster strikes. This feature permits B programs to use TSS subsystems and other programs to do major functions without bookkeeping or core overhead.

We can also use `PRINTF` (→ Appendix C) to write on the `"system"` unit, which is predefined as `-1`. All output written on unit `-1` acts as if typed at command level. Thus,

```
savwunit = .write(-1);
fname = "file";
printf("jrun %s*n", fname);
```

will set the current output unit to -1, the system, and then print on it using `printf` to format the string. This example does the same thing as the last, but the file name is a variable, which can be changed as desired. Notice that this time a `'*n'` is necessary to terminate the line.

One can also say, of course:

```
printf(-1,"jrun %s*n", fname);
```

to effect a temporary change of output unit. Usually this is not necessary, since, apart from adding a terminating `'*n'`, the `SYSTEM` function processes its arguments in the same manner as `printf`.

Because of serious design failings in TSS, there is generally no decent way to send more than the command line as input to a program called this way, and no way to retrieve output from it, except via user-designated files.

31. Recursion

```
putnumb(n) {
    auto a;
    if(a=n/10) /* assignment# not equality test */
                /* i.e. if((a=n/10) != 0) */
        putnumb(a);       /* recursive */
        putchar(n%10 + '0');
}
```

This simplified version of the library function putnumb illustrates the use of recursion. (It only works for `n>0`.) `"a"` is set to the quotient of `n/10`; if this non-zero, a recursive call on `putnumb` is made to print the quotient. Eventually the high-order digit of `n` will be printed, and then as each invocation of `putnumb` is popped up, the next digit will be added on.

32. Argument Passing by functions

There is a subtlety in function usage which can trap the unsuspecting Fortran programmer. Argument passing is "call by value", which means that the called function is given the actual values of its arguments, and doesn't know their addresses. This makes it very hard to change the value of one of the input arguments!

For instance, consider a function `flip(x,y)` which is to interchange its arguments. We might naively write

```
flip(x,y){
    auto t;
    t = x;
    x = y;
    y = t;
}
```

but this has no effect at all. (Non-believers: try it!) What we have to do is pass the arguments as explicit addresses, and use them as addresses. Thus we invoke flip as

```
flip(&a,&b);
```

and the definition of flip is

```
flip(x,y){
    auto t;
    t = *y;
    *x = *y;
    *y = t;
}
```

(And don't leave out the spaces on the right of the equal signs.) This problem doesn't arise if the arguments are vectors, since a vector is represented by its address.

33. Program Arguments : `main(argc,argv)`

An important feature of B not mentioned yet is the fact that when the `main` function itself is called, it is passed two arguments. By tradition, these are usually declared by the programmer as `"argc"` and `"argv"`, for "argument count" and "argument vector", although, of course, any names could actually be used. What these arguments basically transmit is the "command line" with which the program was called. The routine which actually calls your `"main"` function breaks this line up into distinct "arguments" as delimited by tabs and spaces. The number of these arguments is placed in `"argc"`, and `"argv"` is set up as a pointer to an array of pointers to the arguments themselves (usually used like an array of strings).

For example, the command line:

```
go -v srcfil dstfil
```

would cause

```
    argc to contain 4
    argv[0] to point to the string "go"
    argv[1] to point to the string "-v"
    argv[2] to point to the string "srcfil"
and argv[3] to point to the string "dstfil"
```

Note that the last meaningful `"argv"` is always one less than the value in `argc`, since we start with `argv[0]`, although `argv[argc]` is always set to the value `-1`.

As an example, the following is an `"echo"` program written in B; it will `"echo"` its arguments on one line (leaving out the calling program name).

```
main ( argc,argv )
{
    auto i;
    i = 1;
    while(i < argc)
        printf( "%s ",argv[i++] );
    putchar('*n');
}
```

`"argc"` and `"argv"` are a very convenient way to have the user pass information to the program. For instance, we can make use of this in conjunction with the `fcopy` function we showed in the section on 10 (→ 29), to build a program that will allow us to say

```
command file1 file2
```

and have file1 copied to file2. The main could be written:

```
main(argc,argv) {
    if(argc < 3) {
        printf("usage: %s infile outfile*n",argv[0]);
        exit();
    } else
        fcopy(argv[1],argv[2]);
}
```

Reading the command line saves bombarding the user with idiotic prompts such as `"Input file ?"`, `"Output file ?"`, etc. B also relieves the programmer of the responsibility of parsing his own command line, although he can do this if he wants, by calling the library function `"reread"` to read the command line. (See the Reference Manual and b library explain files).

34. Interacting with Fortran and GMAP

As we mentioned, there are certain things such as floating-point and multi-dimensional arrays that Fortran does better than B. A B function, `callf`, allows B programs to call Fortran and GMAP subroutines, like this:

```
callf(name,a1,a2,...,a10)
```

Here name is the Fortran function or subroutine to be used, and the a's are the zero or more arguments of the subroutine.

There are some restrictions on `callf`, related to the "call by value" problem mentioned above. First, variables that are not vectors or strings must be referenced by address: use `"&x"` instead of `"x"` as an argument, as must the Fortran subroutine entry point itself Second, no constants are allowed among the arguments, because the construction `"&constant"` is illegal in B. We can't say

```
tod = callf(&itimet,&0)
```

to get the time of day; instead we have to use

```
t = 0
tod = callf(&itimet,&t)
```

Third, a different B function, `"callff"` must be used, rather than `"callf"` for calling Fortran functions which return float ing-point values. as in

```
s = callff(&sin,&x)
```

Fourth, `calif` and `callff` aren't blindingly efficient, since there are a lot of overhead converting calling sequences. (Very short GMAP routines to interface with B programs can often be coded quite efficiently - see "B Reference Manual", section 6).

References

[1] Johnson/ S. C. A User's Reference to B on MH-TSS.
Bell Laboratories memorandum.

[2] Canaday, R. H., Ritchie, D. M. Bell Laboratories
BCPL. Bell Laboratories memorandum.

There is also documentation on the "B" command in the TSS "EXPLAIN" files. Type "Explain B Index" at command level for a list of available on-line information. This is also the way to obtain your own copy of the "B Reference Manual", by R. P. Gurd, which is currently the most complete documentation available for Waterloo B.

Appendix A : B Compiler Diagnostics

This is a list of diagnostics known to be generated by the B compiler. There may be others.

In each description, "nn" means a line number, while "name" is some identifier name. The name of the source file is usually also given.

Any message not preceded by "warning: " is a fatal error. If there is a fatal error, neither the loader nor the random library editor will be called.

Diagnostics:

`syntax error at line nn [in file <name>]`

This is the most common diagnostic and it could mean almost any kind of error. Most often, it means a semicolon is missing or the number of open curly braces `"{"` does not match the number of close curly braces `"}"`, in which case the line number will be the number of the last line in the last file being processed plus one. This may be due to neglecting to end a string constant, character constant or comment. You also get this message if you use a keyword in an inappropriate context, such as an `AUTO` statement, if you neglect to define a manifest, or if you attempt to redefine a manifest.

`<identifier> undefined in function <name>`

An identifier in the named function has not been referenced by an `EXTRN` or `AUTO` statement and has not been used as a label. The line number given is the last line of the function being compiled.

`warning: /* inside comment ...`

This is a warning only, but there will probably be a syntax error later on, since comments may not be nested. After reading a `"/*"`, the compiler skips all text until a `"*/"` is encountered; if there is a comment inside a comment, then the compiler will attempt to compile the remainder of the outside comment.

`end of file in comment`

This usually indicates that you forgot to end a comment with the terminating `"*/"`.

`warning: newline in constant not preceded by '*'`

The most probable cause is that you forgot to terminate a string or character constant with the appropriate delimiter. If this is the case, you will surely get a syntax error later. If you want a "real'' newline inside the constant, but no warning, use the escape sequence `'*n'`. If the constant is a string constant which is too long to fit on one line, precede the newline with a `'*'`; the newline will be discarded. When the warning is issued, the newline is kept.

`invalid octal constant`

An integer beginning with the digit zero, which is thus assumed to be an octal constant, contains a character other than the digits zero through seven.

`character constant too long`

A character constant may not contain more than four characters, although each character may be a two character escape sequence.

`bed constant too long`

A BCD constant contains more than six characters.

`exponent too large in constant`

The exponent of a floating point constant is too large or too small to represent in the hardware.

`attempt zero division`

In evaluating the constant part of an expression, the right operand of a division or remainder operator was found to be the constant zero.

invalid & prefix
The "*&" operator has been used in an invalid context, such as "&x = y".

warning: found ++r-value
warning: found --r-value
You get this if you say something like "++x++".

invalid $ escape sequence
An escape sequence beginning with '$' is not known to the compiler.

invalid unary operator
The compiler discovered you trying to use a binary operator in a unary manner.

invalid =
invalid *=
invalid >>=
The expression on the left hand side of an assignment operator does not have an lvalue.

invalid ++
invalid --
The expression operated upon by the '++' or operator does not have an lvalue.

invalid label
A name used as a label has previously been declared as EXTRN or AUTO in the current function.

invalid break
The compiler found a BREAK statement which was not inside a FOR, WHILE, DO-WHILE, REPEAT or SWITCH statement.

invalid next
The compiler found a NEXT statement which was not inside a FOR, WHILE, DO-WHILE or REPEAT statement.

invalid constant expression
Will happen if you try to use a string constant in a constant expression.

invalid operator
This is one of those "cannot happen" messages. If it does happen, please submit an error report.

auto array too large
You attempted to declare an auto vector with a dimension greater than 1000 words. It is better to use an external vector or else GETVEC the space, since AUTO variables are allocated on the stack and stack space is limited.

extrn array too large
This will happen if you declare an external vector like "x[3.0];".

invalid case
A CASE label is not inside a SWITCH statement.

invalid default
A DEFAULT label is not inside a SWITCH statement.

default already supplied
: More than one `DEFAULT` label in a `SWITCH` statement.

invalid case operator
: The only bound operators permitted in a CASE are `<`, `>`, `>=` , and `<=`.

%filename ignored - too many open files
: This usually happens when you include a file which includes itself.

bad input character: <ddd> (octal)
: A character encountered in the input stream outside of a string or character constant has no meaning for the compiler. This might be a backspace or some control character typed in by mistake. Since it may be non-printing* the value of the offending character is displayed in octal.

rewrite this expression
: A subscripting expression is too involved for the code generator to handle. Try breaking up the expression into more than one statement.

manifest nesting too deep
: This will occur when you have manifest constants whose evaluation involves other manifest constants. This will occur if you have a series of manifest definitions, each of which is defined in terms of the previous manifest. This is ok in GMAP, but not in B.

warning: program size > 32k
: One of the object decks generated will require more than 32K words to load. You may get this warning if you declare several very large external vectors. However* it might also mean the loader will be aborted by TSS due to "not enough core to run job".

expression too complex
no tree space
no stack space
: An expression is too complex for the compiler to evaluate. Try simplifying it by breaking it up into two or more expressions.

The constant <ddd> occurs in two case labels
: The same constant appears in more than one `CASE` label in a `SWITCH` statement. The value of the offending constant is printed in decimal.

the upper range <ddd> overlaps the lower range <ddd>
: The compiler has detected overlapping bounds inside a `SWITCH` statement. The values of the bounds are displayed in decimal.

The constant <ddd> is in the range <ddd>::<ddd>
: The compiler has detected a `CASE` constant which is in the range of a range case or relational case, in a `SWITCH` statement. The numbers are given in decimal. If something conflicts with a relational case, then the bounds generated for the relation are shown. For example, the bounds for `"case > 0:"` would be `"1::34359738367"`.

Initializers nested too deeply
: An external declaration has initializers in braces nested to a depth greater than seven.

external redefined
auto variable redefined
label redefined
auto array name redefined
: The compiler has detected an attempt to redefine a symbol which has already been defined to the current function body.

no space for symdef
: There are too many external definitions; try dividing them into two groups by either compiling them separately or placing a function in between. This error is almost never encountered.

no space for symref
: There are too many external references in a function definition; try simplification. This error is almost never encountered.

warning: #<text> ignored
: A line beginning with a "#" which is taken to be a compiler directive, does not contain a recognizable directive. The line is ignored.

TSS loader warning messages:
<w> name undefined
: This is a loader message, which indicates that an external variable referenced by one of your functions, or a library function, remains undefined after all libraries have been searched. If your program references the named external it will abort with a MME fault in TSS, or with a USER'S L1 MME GEBORT in batch.

<w> name loaded previously
: The loader has discovered a function or external with the same name as one already loaded. The most probable reason is that you have two or more different names which, when truncated to six characters end up being the same. The loader ignores all but the first. Make sure all your externals and function names are unique in their first six characters.

(Copyright (c) 1979, University of Waterloo)

Appendix B : Escape Sequences and Typography

Description:

There are two sets of escape sequences, one for use inside string or character constants, and the other for use outside.

1. Escape sequences are used in character constants and strings to obtain characters which for one reason or another are hard to represent directly. Here are the escapes:

```
*0      end of string (ASCII NUL = 000)
*e      end of string (ASCII NUL - 000)
*(       { - left curly brace
*)      } - right curly brace
*<      [ - left square bracket
*>      ] - right square bracket
*t      tab
*       *
*'      '
*"      "
*n      newline
*r      carriage return (no line feed)
*f      ASC11 formfeed
*b      backspace
*v      vertical tab
*x      rubout (octal 177)
*#nnn   nnn is 1-3 character octal number
```

2. The following are escapes used outside character and string constants on terminals (such as the 2741) which do not have on their keyboards some of the characters used by B, If you use QED, it is nicer to use the QED escapes for these characters, so that when you shift to an ASCII terminal you can see the characters the way they ought to appear.

```
$(       { - left curly brace
$)       } - right curly brace
$<       [ - left square bracket
$>       ] - right square bracket
$+       | - or-bar
$-       ¢ - up-arrow (or cent-sign)
$a       @ - at-sign
$'       ' - grave accent
```

(Copyright (c) 1979, University of Waterloo)

Appendix C : printf for Formatted IO

`printf([unit]/fmt/a1/a2/a3/.../a10)`

This function writes the arguments ai (from 0 to 10 of which may be present) onto the current output stream under control of the format string `fmt`. The format conversion is controlled by two-letter sequences of the form `'%x'` inside the string `fmt`, where `x` stands for one of the following:

```
c - - character data (ascii)
d - - decimal number
o - - octal number
s - - character string
```

The characters in `fmt` which do not appear in one of these two character sequences are copied without change to the output.

Thus the call

```
printf("%d + %o is %s or %c*n", 1,-1,"zero",'0');
```

leads to the output line

```
1 + 777777777777 is zero or 0
```

The argument `"unit"` is, as indicated, optional, and causes `printf` to do a temporary change of output units. As an example:

```
printf(4,"this is an output line*n");
```

would cause the output to be sent to the device currently associated with unit number `"4"` (which should have been opened for writing prior to the `printf` call).

Appendix D : Binding Strength of Operators

Description:

Operators are listed from highest to lowest binding strength; there is no order within groups. Operators of equal strength bind left to right or right to left as indicated.

```
name   const   primary[expr]   primary(arglist)   (expr)   [LR]
++   --   *   &   -   !   ~   #-   #   ##   (unary)   [RL]
>>  << [LR]
&   [LR]
^   [LR]
|   [LR]
*   /   %   #*   #/   (binary)         [LR]
+  -  #-   #+   [LR]
==   !=   >   <   <=   >=   #==   #*=   #>   #<   #<=   #==   #!=   #>=
&&
||
?:         [RL]
=   =+   =-   etc.   (all assignment operators)   [RL]
```

(Copyright (c) 1979, University of Waterloo)

Appendix E : Index

Name	Section
.read (file IO)	29
.write (file IO)	29
addressing	22
address operator (&)	22
arguments	2,32,33
arithmetic	4,5
arithmetic operators	5
assignment expr'ns	14
assignments, multiple	1
assignment operators	19
auto declaration	4
binding strengths	D
bit operators	18
B (compiling B)	3
callf (call Fortran)	34
case	26
char	23
characters ('…')	6
close	29
comments (/*...*/)	11
compiling B programs	3
compound stat ({...})	12
concat	25
conditionals	11,14,16,2
conditional expr (?:)	16
declarations	4
default	26
diagnostics (errors)	3,A
else	16
escape sequence (*x)	B
exit	15
external vectors	21
extrn declaration	7
fiIe IO	29
formatted IO (printf)	C
Fortran	34
functions	3,8,13,24,31
function values	24
getchar	9
getstr	25
GMAP	34
goto	11
if	11
index	E
indirection op (*)	22
IO (file)	29
IO (formatted)	C
IO (character)	6,9
IO (string)	25
labels	11
layout of B programs	2
lchar	23
logical ops (&&, ||)	27
main	2,33
newline (*n)	6,B
null statement	15
octal numbers	5
open	29
printf	C
program arguments	33
putchar	6
putnumb	4,31
putstr	25
recursion	31
redirection ("<",">")	29
relational ops (==,...)	10
reread	29
return	24
running B programs	3
statement format	2,4
strings ("...")	33
subscripts	20,22
switch	26
system	30
TSS system call	30
types (typeless)	4
unary operators (++, --)	17
variable names (syntax)	4
vectors	20,21
while	14,15